

SpInside MacintoshTM

An on-line development prototype of
Apple's *Inside Macintosh* Volumes I-V

Stack Version: 1.0 • November 1989
HyperCard Version: 2.4.1
System Version: 8.96
Monitors Setting: 8-bit, Color



Apple
Developer Group

Copyright © 1985-1989 by Apple Computer, Inc.

THE SCRAP MANAGER

About This Chapter
About the Scrap Manager
Memory and the Desk Scrap
Desk Scrap Data Types
Using the Scrap Manager
Scrap Manager Routines
 Getting Desk Scrap Information
 Keeping the Desk Scrap on the Disk
 Writing to the Desk Scrap
 Reading from the Desk Scrap
Private Scraps
Format of the Desk Scrap
Summary of the Scrap Manager

ABOUT THIS CHAPTER

This chapter describes the Scrap Manager, the part of the Toolbox that supports cutting and pasting among applications and desk accessories. You should already be familiar with:

- resources, as discussed in the Resource Manager chapter
- QuickDraw pictures
- the Toolbox Event Manager

ABOUT THE SCRAP MANAGER

The Scrap Manager is a set of routines and data types that let Macintosh applications support cutting and pasting using the desk scrap. The desk scrap is the vehicle for transferring data between two applications, between an application and a desk accessory, or between two desk accessories; it can also be used for transferring data that's cut and pasted within an application.

From the user's point of view, all data that's cut or copied resides in the Clipboard. The Cut command deletes data from a document and places it in the Clipboard; the Copy command copies data into the Clipboard without deleting it from the document. The next Paste command—whether applied to the same document or another, in the same application or another—inserts the contents of the Clipboard at a specified place. An application that supports cutting and pasting may also provide a Clipboard window for displaying the current contents of the scrap; it may show the Clipboard window at all times or only when requested via the toggled command Show (or Hide) Clipboard.

Note: The Scrap Manager was designed to transfer small amounts of data; attempts to transfer very large amounts of data may fail due to lack of memory.

The nature of the data to be transferred varies according to the application. For example, in a word processor or in the Calculator desk accessory, the data is text; in a graphics application it's a picture. The amount of information retained about the data being transferred also varies. Between two text applications, text can be cut and pasted without any loss of information; however, if the user of a graphics application cuts a picture consisting of text and then pastes it into a word processor document, the text in the picture may not be editable in the word processor, or it may be editable but not look exactly the same as in the graphics application. The Scrap Manager allows for a variety of data types and provides a mechanism whereby applications have some control over how much information is retained when data is transferred.

The desk scrap is usually stored in memory, but can be stored on the disk (in the scrap file) if there's not enough room for it in memory. The scrap may remain on the disk throughout the use of the application, but must be read back into memory when the application terminates, since the user may then remove that disk and insert another. The Scrap Manager provides routines for writing the desk scrap to the disk and for reading it back into memory. The routines that access the scrap keep track of whether it's in memory or on the disk.

The desk scrap is now written on the system startup volume (that is, the volume that contains the currently open System file) rather than the default volume. With hierarchical volumes, the scrap file is placed in the folder containing the currently open System file and Finder.

In addition, the GetScrap and PutScrap functions will never return the result code noScrapErr; if the scrap has not been initialized, the ZeroScrap function will be called. The InfoScrap function also calls ZeroScrap if the scrap is uninitialized.

MEMORY AND THE DESK SCRAP

The desk scrap is initially located in the application heap; a handle to it is stored in low memory. When starting up an application, the Segment Loader temporarily moves the scrap out of the heap into the stack, reinitializes the heap, and puts the scrap back in the heap (see Figure 1). For a short time while it does this, two copies of the scrap exist in the memory allocated for the stack and the heap; for this reason, the desk scrap cannot be bigger than half that amount of memory.

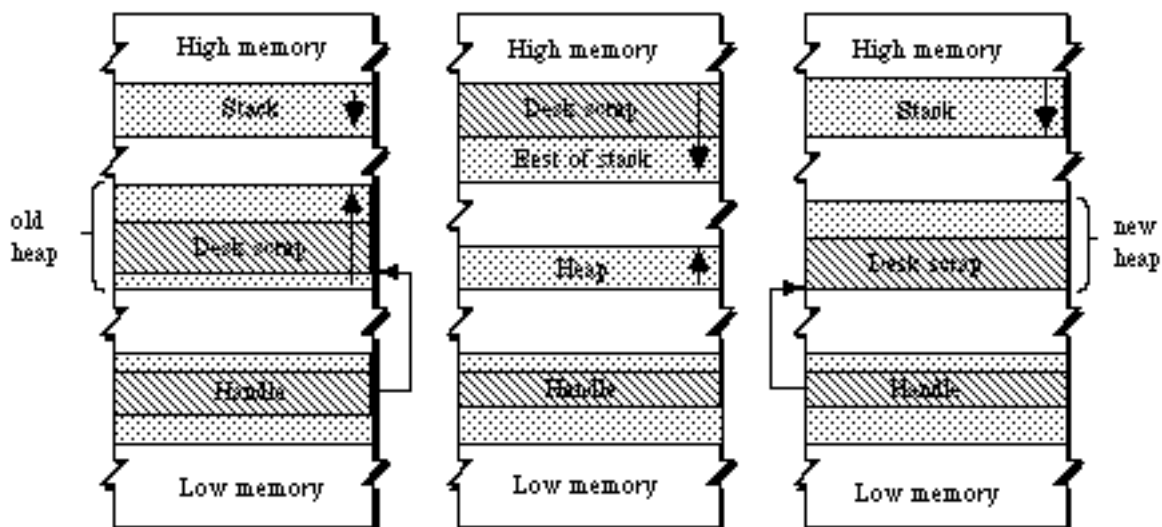


Figure 1—The Desk Scrap at Application Startup

Figure 1—The Desk Scrap at Application Startup

The application can get the size of the desk scrap by calling a Scrap Manager function named InfoScrap. An application concerned about whether there's room for the desk scrap in memory could be set up so that a small initial segment of the application is loaded in just to check the scrap size. After a decision is made about whether to keep the scrap in memory or on the disk, the remaining segments of the application can be loaded in as needed.

There are certain disadvantages to keeping the desk scrap on the disk. The disk may be locked, it may not have enough room for the scrap, or it may be removed during use of the application. If the application can't write the scrap to the disk, it should put up an alert box informing the user, who may want to abort the operation at that point.

DESK SCRAP DATA TYPES

From the user's point of view there can be only one thing in the Clipboard at a time, but the application may store more than one version of the information in the scrap, each representing the same Clipboard contents in a different form. For example, text cut with a word processor may be stored in the desk scrap both as text and as a QuickDraw picture.

Desk scrap data types, like resource types, are a sequence of four characters. As defined in the Resource Manager, their Pascal type is as follows:

```
TYPE ResType = PACKED ARRAY[1..4] OF CHAR;
```

Two standard types of data are defined:

- 'TEXT': a series of ASCII characters
- 'PICT': a QuickDraw picture, which is a saved sequence of drawing commands that can be played back with the DrawPicture command and may include picture comments (see the QuickDraw chapter for details)

Applications must write at least one of these standard types of data to the desk scrap and must be able to read both types. Most applications will prefer one of these types over the other; for example, a word processor prefers text while a graphics application prefers pictures. An application should write at least its preferred standard type of data to the desk scrap, and may write both types (to pass the most information possible on to the receiving application, which may prefer the other type).

An application reading the desk scrap will look for its preferred data type. If its preferred type isn't there, or if it's there but was written by an application having a different preferred type, the receiving application may or may not be able to convert the data to the type it needs. If not, some information may be lost in the transfer process. For example, a graphics application can easily convert text to a picture, but the reverse isn't true. Figure 2 illustrates the latter case: A picture consisting of text is cut from a graphics application, then pasted into a word processor document.

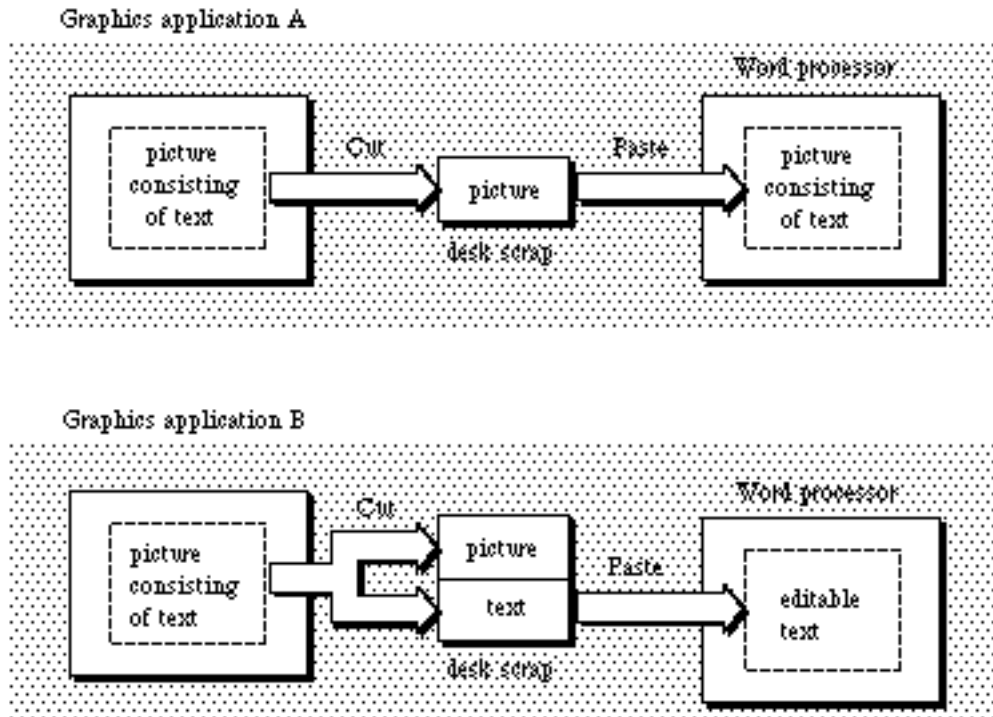


Figure 2-Inter-Application Cutting and Pasting

Figure 2-Inter-Application Cutting and Pasting

- If the graphics application writes only its preferred data type (picture) to the desk scrap—like application A in Figure 2—the text in the picture will not be editable in the word processor, because it will be seen as just a series of drawing commands and not as a sequence of characters.
- On the other hand, if the graphics application takes the trouble to recognize which characters have been drawn in the picture, and writes them out to the desk scrap both as a picture and as text—like application B in Figure 2—the word processor will be able to treat them as editable text. In this case, however, any part of the picture that isn't text will be lost.

In addition to the two standard data types, the desk scrap may also contain application-specific types of data. If several applications are to support the transfer of a private type of data, each one will write and read that type, but still must write at least one of the standard types and be able to read both standard types.

The order in which data is written to the desk scrap is important: The application should write out the different types in order of preference. For example, if it's a word processor that has a private type of data as its preferred type, but also can write text and pictures, it should write the data in that order.

Since the size of the desk scrap is limited, it may be too costly to write out both an application-specific data type and one (or both) of the standard types.

Instead of creating your own type, if your data is graphic, you may be able to use the standard picture type and encode additional information in picture comments. (As described in the QuickDraw chapter, picture comments may be stored in the definition of a picture with the QuickDraw procedure `PicComment`; they're passed by the `DrawPicture` procedure to a special routine set up by the application for that purpose.) Applications that are to process that information can do so, while others can ignore it.

USING THE SCRAP MANAGER

If you're concerned about memory use, call InfoScrap early in your program to find out the size of the desk scrap and decide whether there will be enough room in the heap for both the desk scrap and the application itself. If there won't be enough room for the scrap, call UnloadScrap to write the scrap from memory onto the disk.

InfoScrap also provides a handle to the desk scrap if it's in memory, its file name on the disk, and a count that changes when the contents of the desk scrap change. If your application supports display of the Clipboard, you can call InfoScrap each time through your main event loop to check this count: If the Clipboard window is visible, it needs to be updated whenever the count changes.

When a Cut or Copy command is given, you need to write the cut or copied data to the desk scrap. First call ZeroScrap to initialize the scrap or clear its previous contents, and then PutScrap to put the data into the scrap. (You can call PutScrap more than once, to put the data in the scrap in different forms.)

Call GetScrap when a Paste command is given, to access data of a particular type in the desk scrap and to get information about the data.

When the user gives a command that terminates the application, call LoadScrap to read the desk scrap back into memory if it's on the disk (in case the user ejects the disk).

Note: ZeroScrap, PutScrap, and GetScrap all keep track of whether the scrap is in memory or on the disk, so you don't have to worry about it.

If your application uses TextEdit and the TextEdit scrap, you'll need to transfer data between the two scraps, as described in the section "Private Scraps", below.

SCRAP MANAGER ROUTINES

Most of these routines return a result code indicating whether an error occurred. If no error occurred, they return the result code

CONST noErr = 0; {no error}

If an error occurred at the Operating System level, an Operating System result code is returned; otherwise, a Scrap Manager result code is returned, as indicated in the routine descriptions. (See Appendix A for a list of all result codes.)

Getting Desk Scrap Information

FUNCTION InfoScrap : PScrapStuff;

InfoScrap returns a pointer to information about the desk scrap. The PScrapStuff data type is defined as follows:

```
TYPE PScrapStuff = ^ScrapStuff;
     ScrapStuff = RECORD
         scrapSize:    LONGINT;    {size of desk scrap}
         scrapHandle:  Handle;     {handle to desk scrap}
         scrapCount:   INTEGER;    {count changed by ZeroScrap}
         scrapState:   INTEGER;    {tells where desk scrap is}
         scrapName:    StringPtr  {scrap file name}
     END;
```

ScrapSize is the size of the desk scrap in bytes. ScrapHandle is a handle to the scrap if it's in memory, or NIL if not.

ScrapCount is a count that changes every time ZeroScrap is called. You can use this count for testing whether the contents of the desk scrap have changed, since if ZeroScrap has been called, presumably PutScrap was also called. This may be useful if your application supports display of the Clipboard or has a private scrap (as described under "Private Scraps", below).

Warning: Just check to see whether the value of the scrapCount field has changed; don't rely on exactly how it has changed.

ScrapState is positive if the desk scrap is in memory, 0 if it's on the disk, or negative if it hasn't been initialized by ZeroScrap.

Note: ScrapState is actually 0 if the scrap should be on the disk; for instance, if the user deletes the Clipboard file and then cuts something, the scrap is really in memory, but ScrapState will be 0.

ScrapName is a pointer to the name of the scrap file, usually "Clipboard File".

Note: InfoScrap assumes that the scrap file has a version number of 0 and is on the default volume. (Version numbers and volumes are described in the File Manager chapter.)

Assembly-language note: The scrap information is available in global variables that have the same names as the Pascal fields.

Keeping the Desk Scrap on the Disk

FUNCTION UnloadScrap : LONGINT;

Assembly-language note: The macro you invoke to call UnloadScrap from assembly language is named `_UnloadScrap`.

UnloadScrap writes the desk scrap from memory to the scrap file, and releases the memory it occupied. If the desk scrap is already on the disk, UnloadScrap does nothing. If no error occurs, UnloadScrap returns the result code `noErr`; otherwise, it returns an Operating System result code indicating an error.

FUNCTION LoadScrap : LONGINT;

Assembly-language note: The macro you invoke to call LoadScrap from assembly language is named `_LoadScrap`.

LoadScrap reads the desk scrap from the scrap file into memory. If the desk scrap is already in memory, it does nothing. If no error occurs, LoadScrap returns the result code `noErr`; otherwise, it returns an Operating System result code indicating an error.

Writing to the Desk Scrap

FUNCTION ZeroScrap : LONGINT;

If the scrap already exists (in memory or on the disk), ZeroScrap clears its contents; if not, the scrap is initialized in memory. You must call ZeroScrap before the first time you call PutScrap. If no error occurs, ZeroScrap returns the result code `noErr`; otherwise, it returns an Operating System result code indicating an error.

ZeroScrap also changes the `scrapCount` field of the record of information provided by InfoScrap.

FUNCTION PutScrap (length: LONGINT; theType: ResType;
 source: Ptr) : LONGINT;

PutScrap writes the data pointed to by the source parameter to the desk scrap (in memory or on the disk). The length parameter indicates the number of bytes to write, and theType is the data type.

Warning: The specified type must be different from the type of any data already in the desk scrap. If you write data of a type already in

the scrap, the new data will be appended to the scrap, and subsequent GetScrap calls will still return the old data.

If no error occurs, PutScrap returns the result code noErr; otherwise, it returns an Operating System result code indicating an error, or the following Scrap Manager result code:

```
CONST noScrapErr = -100; {desk scrap isn't initialized}
```

Warning: Don't forget to call ZeroScrap to initialize the scrap or clear its previous contents.

Note: To copy the TextEdit scrap to the desk scrap, use the TextEdit function TEToScrap.

Reading from the Desk Scrap

```
FUNCTION GetScrap (hDest: Handle; theType: ResType;  
                  VAR offset: LONGINT) : LONGINT;
```

Given an existing handle in hDest, GetScrap reads the data of type theType from the desk scrap (whether in memory or on the disk), makes a copy of it in memory, and sets hDest to be a handle to the copy. Usually you'll pass in hDest a handle to a minimum-size block; GetScrap will resize the block and copy the scrap into it. If you pass NIL in hDest, GetScrap will not read in the data. This is useful if you want to be sure the data is there before allocating space for its handle, or if you just want to know the size of the data.

In the offset parameter, GetScrap returns the location of the data as an offset (in bytes) from the beginning of the desk scrap. If no error occurs, the function result is the length of the data in bytes; otherwise, it's either an appropriate Operating System result code (which will be negative) or the following Scrap Manager result code:

```
CONST noTypeErr = -102; {no data of the requested type}
```

For example, given the declarations

```
VAR pHndl: Handle;    {handle for 'PICT' type}  
    tHndl: Handle;    {handle for 'TEXT' type}  
    length: LONGINT;  
    offset: LONGINT;  
    frame: Rect;
```

you can make the following calls:

```
pHndl := NewHandle(0);  
length := GetScrap(pHndl, 'PICT', offset);  
IF length < 0  
    THEN  
        {error handling}  
    ELSE DrawPicture(PicHandle(pHndl), frame)
```

If your application wants data in the form of a picture, and the scrap contains only text, you can convert the text into a picture by doing the following:

```
tHndl := NewHandle(0);
length := GetScrap(tHndl, 'TEXT', offset);
IF length < 0
    THEN
        {error handling}
    ELSE
        BEGIN
            HLock(tHndl);
            pHndl := OpenPicture(thePort^.portRect);
            TextBox(tHndl^, length, thePort^.portRect, teJustLeft);
            ClosePicture;
            HUnlock(tHndl);
        END
```

The Memory Manager procedures HLock and HUnlock are used to lock and unlock blocks when handles are dereferenced (see the Memory Manager chapter).

Note: To copy the desk scrap to the TextEdit scrap, use the TextEdit function TEFromScrap.

Your application should pass its preferred data type to GetScrap. If it doesn't prefer one data type over any other, it should try getting each of the types it can read, and use the type that returns the lowest offset. (A lower offset means that this data type was written before the others, and therefore was preferred by the application that wrote it.)

Note: If you're trying to read in a complicated picture, and there isn't enough room in memory for a copy of it, you can customize QuickDraw's picture retrieval so that DrawPicture will read the picture directly from the scrap file. (QuickDraw also lets you customize how pictures are saved so you can save them in a file; see the QuickDraw chapter for details about customizing.)

Note: When reading in a picture from the scrap, allow a buffer of about 3.5K bytes on the stack.

PRIVATE SCRAPS

Note: MultiFinder keeps separate scrap variables for each partition. For more information about managing private scraps with MultiFinder running, refer to the "Programmer's Guide to MultiFinder" and Technical Notes #180 and #205.

•••Click on the X-Ref button, and refer to Technical Note #180 & #205.•••

Instead of using the desk scrap for storing data that's cut and pasted within an application, advanced programmers may want to set up a private scrap for this purpose. In applications that use the standard 'TEXT' or 'PICT' data types, it's simpler to use the desk scrap, but if your application defines its own private type of data, or if it's likely that very large amounts of data will be cut and pasted, using a private scrap may result in faster cutting and pasting within the application.

The format of a private scrap can be whatever the application likes, since no other application will use it. For example, an application can simply maintain a pointer to data that's been cut or copied. The application must, however, be able to convert data between the format of its private scrap and the format of the desk scrap.

Note: The TextEdit scrap is a private scrap for applications that use TextEdit. TextEdit provides routines for accessing this scrap; you'll need to transfer data between the TextEdit scrap and the desk scrap.

If you use a private scrap, you must be sure that the right data will always be pasted when the user gives a Paste command (the right data being whatever was most recently cut or copied in any application or desk accessory), and that the Clipboard, if visible, always shows the current data. You should copy the contents of the desk scrap to your private scrap at application startup and whenever a desk accessory is deactivated (call GetScrap to access the desk scrap). When the application is terminated or when a desk accessory is activated, you should copy the contents of the private scrap to the desk scrap: Call ZeroScrap to initialize the desk scrap or clear its previous contents, and PutScrap to write data to the desk scrap.

If transferring data between the two scraps means converting it, and possibly losing information, you can copy the scrap only when you actually need to, at the time something is cut or pasted. The desk scrap needn't be copied to the private scrap unless a Paste command is given before the first Cut or Copy command since the application started up or since a desk accessory that changed the scrap was deactivated. Until that point, you must keep the contents of the desk scrap intact, displaying it instead of the private scrap in the Clipboard window if that window is visible. Thereafter, you can ignore the desk scrap until a desk accessory is activated or the application is terminated; in either of these cases, you must copy the private scrap back to the desk scrap. Thus whatever was last cut or copied within the application will be pasted if a Paste command is given in a desk accessory or in the next application. If no Cut or

Copy commands are given within the application, you never have to change the desk scrap.

To find out whether a desk accessory has changed the desk scrap, you can check the scrapCount field of the record returned by InfoScrap. Save the value of this field when one of your application's windows is deactivated and a system window is activated. Check each time through the main event loop to see whether its value has changed; if so, the contents of the desk scrap have changed.

If the application encounters problems in trying to copy one scrap to another, it should alert the user. The desk scrap may be too large to copy to the private scrap, in which case the user may want to leave the application or just proceed with an empty Clipboard. If the private scrap is too large to copy to the desk scrap, either because it's disk-based and too large to copy into memory or because it exceeds the maximum size allowed for the desk scrap, the user may want to stay in the application and cut or copy something smaller.

FORMAT OF THE DESK SCRAP

In general, the desk scrap consists of a series of data items that have the following format:

Number of bytes	Contents
4 bytes	Type (a sequence of four characters)
4 bytes	Length of following data in bytes
n bytes	Data; n must be even (if the above length is odd, add an extra byte)

The standard types are 'TEXT' and 'PICT'. You may use any other four-character sequence for types specific to your application.

The format of the data for the 'TEXT' type is as follows:

Number of bytes	Contents
4 bytes	Number of characters in the text
n bytes	The characters in the text

The data for the 'PICT' type is a QuickDraw picture, which consists of the size of the picture in bytes, the picture frame, and the picture definition data (which may include picture comments). See the QuickDraw chapter for details.

SUMMARY OF THE SCRAP MANAGER

Constants

CONST

```
{ Scrap Manager result codes }

noScrapErr = -100;    {desk scrap isn't initialized}
noTypeErr = -102;    {no data of the requested type}
```

Data Types

TYPE

```
PScrapStuff = ^ScrapStuff;
ScrapStuff = RECORD
    scrapSize:    LONGINT;    {size of desk scrap}
    scrapHandle:  Handle;     {handle to desk scrap}
    scrapCount:   INTEGER;    {count changed by ZeroScrap}
    scrapState:   INTEGER;    {tells where desk scrap is}
    scrapName:    StringPtr   {scrap file name}
END;
```

Routines

Getting Desk Scrap Information

```
FUNCTION InfoScrap : PScrapStuff;
```

Keeping the Desk Scrap on the Disk

```
FUNCTION UnloadScrap : LONGINT;
FUNCTION LoadScrap :    LONGINT;
```

Writing to the Desk Scrap

```
FUNCTION ZeroScrap : LONGINT;
FUNCTION PutScrap    (length: LONGINT; theType: ResType;
                     source: Ptr) : LONGINT;
```

Reading from the Desk Scrap

```
FUNCTION GetScrap (hDest: Handle; theType: ResType;
                  VAR offset: LONGINT) : LONGINT;
```

Assembly-Language Information

Constants

; Scrap Manager result codes

```
noScrapErr    .EQU    -100    ;desk scrap isn't initialized
noTypeErr     .EQU    -102    ;no data of the requested type
```

Special Macro Names

Pascal name	Macro name
LoadScrap	_LodeScrap
UnloadScrap	_UnlodeScrap

Variables

ScrapSize	Size in bytes of desk scrap (long)
ScrapHandle	Handle to desk scrap in memory
ScrapCount	Count changed by ZeroScrap (word)
ScrapState	Tells where desk scrap is (word)
ScrapName	Pointer to scrap file name (preceded by length byte)

Further Reference:

Resource Manager

QuickDraw

Toolbox Event Manager

Technical Note #180, MultiFinder Miscellanea

Technical Note #205, MultiFinder Revisited: The 6.0 System Release



END OF DOCUMENT